The Stack & Procedures
CSE 351 Spring 2018

http://xkcd.com/648/
Roadmap

C:

```c
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

Java:

```java
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
c.getMPG();
```

Assembly language:

```
get_mpg:
    pushq %rbp
    movq %rsp, %rbp
    ...
    popq %rbp
    ret
```

Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101010000111111
```

OS:

- Windows 10
- OS X Yosemite

Memory & data
Integers & floats
x86 assembly
Procedures & stacks
Executables
Arrays & structs
Processes
Virtual memory
Memory & caches
Memory allocation
Java vs. C
Mechanisms required for procedures

1) Passing control
   - To beginning of procedure code
   - Back to return point

2) Passing data
   - Procedure arguments
   - Return value

3) Memory management
   - Allocate during procedure execution
   - Deallocate upon return

   All implemented with machine instructions!
   - An x86-64 procedure uses only those mechanisms required for that procedure

```c
int Q(int i) {
    int t = 3*i;
    int v[10];
    ...
    return v[t];
}

P(...) {
    ...
    y = Q(x);
    print(y)
    ...
}
```
Procedures

- **Stack Structure**
- **Calling Conventions**
  - Passing control
  - Passing data
  - Managing local data
- **Register Saving Conventions**
- **Illustration of Recursion**
Simplified Memory Layout

High Addresses \[2^{N-1}\]

Memory Addresses

Low Addresses \[0\]

- **Stack**: local variables; procedure context
- **Dynamic Data (Heap)**: variables allocated with `new` or `malloc`
- **Static Data**: static variables (including global variables (C))
- **Literals**: large constants (e.g. “example”)
- **Instructions**: program code
Memory Permissions

- **Stack**
  - Managed “automatically” (by compiler)
  - Writable; not executable

- **Dynamic Data (Heap)**
  - Managed by programmer
  - Writable; not executable

- **Static Data**
  - Initialized when process starts
  - Writable; not executable

- **Literals**
  - Initialized when process starts
  - Read-only; not executable

- **Instructions**
  - Initialized when process starts
  - Read-only; executable

Segmentation faults?
x86-64 Stack

- Region of memory managed with stack “discipline”
  - Grows toward lower addresses
  - Customarily shown “upside-down”

- Register $%rsp$ contains lowest stack address
  - $%rsp$ = address of top element, the most_recently_pushed item that is not_yet_popped

Stack Pointer: $%rsp$
x86-64 Stack: Push

- `pushq src`
  - Fetch operand at `src`
    - `Src` can be reg, memory, immediate
  - *Decrement* `%rsp` by 8
  - Store value at address given by `%rsp`

- **Example:**
  - `pushq %rcx`
  - Adjust `%rsp` and store contents of `%rcx` on the stack

  **Stack Pointer:** `%rsp`
x86-64 Stack: Pop

- `popq dst`
  - Load value at address given by `%rsp`
  - Store value at `dst` (must be register)
  - **Increment** `%rsp` by 8

- **Example**:
  - `popq %rcx`
  - Stores contents of top of stack into `%rcx` and adjust `%rsp`

Those bits are still there; we’re just not using them.
Procedures

- Stack Structure
- **Calling Conventions**
  - Passing control
  - Passing data
  - Managing local data
- Register Saving Conventions
- Illustration of Recursion
Procedure Call Overview

- **Callee** must know where to find args
- **Callee** must know where to find `return address`
- **Caller** must know where to find `return value`
- **Caller** and **Callee** run on same CPU, so use the same registers
  - How do we deal with register reuse?
- Unneeded steps can be skipped (e.g. no arguments)
The convention of where to leave/find things is called the calling convention (or procedure call linkage)

- Details vary between systems
- We will see the convention for x86-64/Linux in detail
- What could happen if our program didn’t follow these conventions?
Code Example (Preview)

```c
void multstore
(long x, long y, long *dest)
{
    long t = mult2(x, y);
    *dest = t;
}
```

```c
long mult2
(long a, long b)
{
    long s = a * b;
    return s;
}
```

Compiler Explorer: [https://godbolt.org/g/cKKDZn](https://godbolt.org/g/cKKDZn)
Procedure Control Flow

- Use stack to support procedure call and return
- **Procedure call**: `call label`
  1) Push return address on stack (*why? which address?*)
  2) Jump to `label`
Procedure Control Flow

- Use stack to support procedure call and return

  **Procedure call:** `call label`
  1) Push return address on stack (*why? which address?*)
  2) Jump to `label`

- Return address:
  - Address of instruction immediately after `call` instruction
  - Example from disassembly:
    
    ```
    400544: callq 400550 <mult2>
    400549: movq %rax,(%rbx)
    ```
    Return address = `0x400549`

- Procedure return: `ret`
  1) Pop return address from stack
  2) Jump to address
Procedure **Call Example** (step 1)

0000000000400540 <multstore>:

- ...
- 400544: call 400550 <mult2>
- 400549: movq %rax, (%rbx)
- ...
- ...

0000000000400550 <mult2>:

- 400550: movq %rdi, %rax
- ...
- 400557: ret
**Procedure Call Example (step 2)**

### `multstore` (step 2)

```
0000000000400540 <multstore>:

400544: call 400550 <mult2>
400549: movq %rax,(%rbx)
```

### `mult2` (step 1)

```
0000000000400550 <mult2>:

400550: movq %rdi,%rax
400557: ret
```
Procedure Return Example (step 1)

00000000000400540 <multstore>:
•
400544: call 400550 <mult2>
400549: movq %rax, (%rbx)
•

00000000000400550 <mult2>:
400550: movq %rdi, %rax
•
400557: ret
Procedure Return Example (step 2)

0000000000400540 <multstore>:

400544: call 400550 <mult2>
400549: movq %rax,(%rbx)

0000000000400550 <mult2>:

400550: movq %rdi,%rax

400557: ret
Procedures

- Stack Structure
- **Calling Conventions**
  - Passing control
  - *Passing data*
  - Managing local data
- Register Saving Conventions
- Illustration of Recursion
Procedure Data Flow

Registers (NOT in Memory)
- First 6 arguments
  - %rdi
  - %rsi
  - %rdx
  - %rcx
  - %r8
  - %r9

- Return value
  - %rax

Stack (Memory)
- Only allocate stack space when needed
x86-64 Return Values

- By convention, values returned by procedures are placed in %rax
  - Choice of %rax is arbitrary

1) **Caller** must make sure to save the contents of %rax before calling a **callee** that returns a value
   - Part of register-saving convention

2) **Callee** places return value into %rax
   - Any type that can fit in 8 bytes – integer, float, pointer, etc.
   - For return values greater than 8 bytes, best to return a *pointer* to them

3) Upon return, **caller** finds the return value in %rax
Data Flow Examples

```c
void multstore
(long x, long y, long *dest)
{
    long t = mult2(x, y);
    *dest = t;
}
```

```assembly
0000000000400550 <multstore>:
    # x in %rdi, y in %rsi, dest in %rdx
    ...
400541:  movq    %rdx,%rbx  # Save dest
400544:  call    400550 <mult2>  # mult2(x,y)
    # t in %rax
400549:  movq    %rax,(%rbx)  # Save at dest
    ...
```

```c
long mult2
(long a, long b)
{
    long s = a * b;
    return s;
}
```

```assembly
0000000000400550 <mult2>:
    # a in %rdi, b in %rsi
400550:  movq    %rdi,%rax  # a
400553:  imulq   %rsi,%rax  # a * b
    # s in %rax
400556:  ret     # Return
```
Procedures

- Stack Structure
- **Calling Conventions**
  - Passing control
  - Passing data
  - **Managing local data**
- Register Saving Conventions
- Illustration of Recursion
Stack-Based Languages

- Languages that support recursion
  - *e.g.* C, Java, most modern languages
  - Code must be **re-entrant**
    - Multiple simultaneous instantiations of single procedure
  - Need some place to store *state* of each instantiation
    - Arguments, local variables, return pointer

- Stack allocated in **frames**
  - State for a single procedure instantiation

- Stack discipline
  - State for a given procedure needed for a limited time
    - Starting from when it is called to when it returns
  - Callee always returns before caller does
Call Chain Example

```
yoo(...) {
  •
  •
  who();
  •
}

who(...) {
  •
  amI();
  •
}

amI(...) {
  •
  if(...) {
    amI()
  }
  •
}
```

Procedure `amI` is recursive (calls itself)
1) Call to `yoo`

```c
yoo (...) {
  •
  •
  who();
  •
}
```

Stack diagram:
```
Stack
  |
  --
  |
  |
  |
yoo
  |
  %rbp
  |
  %rsp
  |
  amI
  §
  |
  amI
  |
  amI
  |
  amI
```
2) Call to who

```
yoo (...) {
  who (...) {
    •
    amI();
    •
    amI();
    •
  }
}
```

Stack

```
yoo
who
amI  amI
  amI
%rbp
%rsp
```
3) Call to `amI` (1)
4) Recursive call to amI (2)
5) (another) Recursive call to `amI (3)`

```plaintext
yoo(…)
{
  who(…)
  {
    amI(…)
    {
      amI(…)
      {
        if()
        {
          amI()
        }
      }
    }
  }
}
```
6) Return from (another) recursive call to \texttt{amI}
7) Return from recursive call to `amI`

```
yoo(...)
{
  who(...)
  {
    amI(...)
    {
      •
      if()
      amI()
    }
    •
  }
}
```
8) Return from call to `amI`
9) **(second) Call to amI (4)**

```
yoo(…)
{
  who(…)
  {
    amI(…)
    {
      •
      if(){
        amI()
      }
    }
  }
  •
}
```

Stack

- yoo
- who
- amI
- amI
- amI
- %rbp
- %rsp
- amI_4
- amI_3
- amI_2
10) Return from (second) call to `amI`

```
yoo(…)
{
  who(…)
  {
    •
    amI();
    •
    amI();
  }
}
```

Stack

```
%rbp
yoo
who
%rsp
amI
amI
amI
amI
```
11) Return from call to `who`

```c
yoo (...) {
    ...
    who();
    ...
}
```

Stack diagram:
- `yoo`
- `%rbp` link
- `%rsp` link
- `who`
- `amI`
- `amI`
x86-64/Linux Stack Frame

- **Caller’s Stack Frame**
  - Extra arguments (if > 6 args) for this call

- **Current/Callee Stack Frame**
  - Return address
    - Pushed by `call` instruction
  - Old frame pointer (optional)
  - Saved register context (when reusing registers)
  - Local variables
    (If can’t be kept in registers)
  - “Argument build” area
    (If callee needs to call another function - parameters for function about to call, if needed)
Peer Instruction Question

Answer the following questions about when `main()` is run (assume `x` and `y` stored on the Stack):

- **Higher/larger address**: `x` or `y`?
- **How many total stack frames are created**?
- **What is the maximum depth (# of frames) of the Stack**?

```c
int main() {
    int i, x = 0;
    for(i=0;i<3;i++)
        x = randSum(x);
    printf("x = %d\n",x);
    return 0;
}
```

```c
int randSum(int n) {
    int y = rand() % 20;
    return n+y;
}
```

A. 1  B. 2  C. 3  D. 4
Example: increment

```c
long increment(long *p, long val) {
    long x = *p;
    long y = x + val;
    *p = y;
    return x;
}
```

increment:

```
movq (%rdi), %rax
addq %rax, %rsi
movq %rsi, (%rdi)
ret
```

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>1st arg (p)</td>
</tr>
<tr>
<td>%rsi</td>
<td>2nd arg (val), y</td>
</tr>
<tr>
<td>%rax</td>
<td>x, return value</td>
</tr>
</tbody>
</table>
Procedure Call Example (initial state)

```c
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

Initial Stack Structure

- Return address on stack is the address of instruction immediately following the call to “call_incr”
  - Shown here as `main`, but could be anything)
  - Pushed onto stack by `call call_incr`
Procedure Call Example (step 1)

```c
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1 + v2;
}
```

- **Setup space for local variables**
  - Only `v1` needs space on the stack
- **Compiler allocated extra space**
  - Often does this for a variety of reasons, including alignment

---

Stack Structure

- Return addr `<main+8>`
- Unused
- 351
- `old %rsp` ← %rsp+8
- `old %rsp` ← %rsp

Allocate space for local vars

- **Setup space for local variables**
  - Only `v1` needs space on the stack
- **Compiler allocated extra space**
  - Often does this for a variety of reasons, including alignment
Procedure Call Example (step 2)

```c
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

```
call_incr:
    subq $16, %rsp
    movq $351, 8(%rsp)
    movl $100, %esi
    leaq 8(%rsp), %rdi
    call increment
    addq 8(%rsp), %rax
    addq $16, %rsp
    ret
```

Stack Structure

```
Return addr <main+8>
351 ←%rsp+8
Unused ←%rsp
```

Set up parameters for call to increment

Aside: movl is used because 100 is a small positive value that fits in 32 bits. High order bits of rsi get set to zero automatically. It takes one less byte to encode a movl than a movq.
Procedure Call Example (step 3)

State while inside `increment`
- **Return address** on top of stack is address of the `addq` instruction immediately following call to `increment`

```
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1 + v2;
}
```

```
call_incr:
    subq $16, %rsp
    movq $351, 8(%rsp)
    movl $100, %esi
    leaq 8(%rsp), %rdi
    call increment
    addq 8(%rsp), %rax
    addq $16, %rsp
    ret
```

```
increment:
    movq (%rdi), %rax
    addq %rax, %rsi
    movq %rsi, (%rdi)
    ret
```

Stack Structure

Return addr `<main+8>`
- 351
  - `Unused`

Return addr `<call_incr+?>` ← %rsp
Procedure Call Example (step 4)

```c
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

**Stack Structure**

- Return addr `<main+8>`
- 451
- Unused
- Return addr `<call_incr+?>`

- State while inside `increment`
  - After code in body has been executed

**Increment:**

```assembly
movq (%rdi), %rax  # x = *p
addq %rax, %rsi  # y = x+100
movq %rsi, (%rdi)  # *p = y
ret
```

**Register Use(s)**

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>&amp;v1</td>
</tr>
<tr>
<td>%rsi</td>
<td>451</td>
</tr>
<tr>
<td>%rax</td>
<td>351</td>
</tr>
</tbody>
</table>
Procedure Call Example (step 5)

```c
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

- After returning from call to `increment`
  - Registers and memory have been modified and return address has been popped off stack

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>&amp;v1</td>
</tr>
<tr>
<td>%rsi</td>
<td>451</td>
</tr>
<tr>
<td>%rax</td>
<td>351</td>
</tr>
</tbody>
</table>

Stack Structure:

- Return addr <main+8>
- 451
- Unused
Procedure Call Example (step 6)

```c
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1 + v2;
}
```

**Call in incr:**
- `subq $16, %rsp`
- `movq $351, 8(%rsp)`
- `movl $100, %esi`
- `leaq 8(%rsp), %rdi`
- `call increment`
- `addq 8(%rsp), %rax`
- `addq $16, %rsp`
- `ret`

**Stack Structure**

<table>
<thead>
<tr>
<th>Register Use(s)</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;v1</td>
<td>%rdi</td>
</tr>
<tr>
<td>451</td>
<td>%rsi</td>
</tr>
<tr>
<td>451+351</td>
<td>%rax</td>
</tr>
</tbody>
</table>

• • •

Return addr <main+8>

451

Unused

← %rsp

← %rsp + 8

Update %rax to contain v1 + v2
Procedure Call Example (step 7)

```c
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1 + v2;
}
```

Stack Structure

- Return addr <main+8>
- %rsp ➜ old %rsp
- 451
- Unused

Stack Structure Diagram:

- De-allocate space for local vars

Register Use(s)

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>&amp;v1</td>
</tr>
<tr>
<td>%rsi</td>
<td>451</td>
</tr>
<tr>
<td>%rax</td>
<td>802</td>
</tr>
</tbody>
</table>
Procedure Call Example (step 8)

```c
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

- **State just before returning from call to call_incr**

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>&amp;v1</td>
</tr>
<tr>
<td>%rsi</td>
<td>451</td>
</tr>
<tr>
<td>%rax</td>
<td>802</td>
</tr>
</tbody>
</table>
Procedure Call Example (step 9)

```c
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1 + v2;
}
```

- State immediately after returning from call to `call_incr`
  - Return addr has been popped off stack
  - Control has returned to the instruction immediately following the call to `call_incr` (not shown here)

**Final Stack Structure**

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>%rdi</code></td>
<td>&amp;v1</td>
</tr>
<tr>
<td><code>%rsi</code></td>
<td>451</td>
</tr>
<tr>
<td><code>%rax</code></td>
<td>802</td>
</tr>
</tbody>
</table>